EXHIBIT PP

U.S. Patent No. 7,519,814 vs. HPE

Accused Instrumentalities: HPE products and services using secure containerized applications, including without limitation HPE's Ezmeral Runtime Enterprise (including without limitation both Ezmeral Runtime Enterprise and Ezmeral Runtime Enterprise Essentials, in each case including when marketed, sold, and/or licensed as part of or associated with ~~and~~ HPE's GreenLake branding, e.g. "HPE GreenLake for containers" which "is built on HPE Ezmeral Container Platform"), and all versions and variations thereof since the issuance of the asserted patent.

Each Accused Instrumentality infringes the claims in substantially the same way, and the evidence shown in this chart is similarly applicable to each Accused Instrumentality. Each claim limitation is literally infringed by each Accused Instrumentality. However, to the extent any claim limitation is not met literally, it is nonetheless met under the doctrine of equivalents because the differences between the claim limitation and each Accused Instrumentality would be insubstantial, and each Accused Instrumentality performs substantially the same function, in substantially the same way, to achieve the same result as the claimed invention. Notably, Defendant has not yet articulated which, if any, particular claim limitations it believes are not met by the Accused Instrumentalities.

Claim 1

Claim 1	Accused Instrumentalities
[1pre] 1. In a system having a plurality of servers with operating systems that differ, operating in disparate computing environments, wherein each server includes a processor and an operating system including a kernel a set of associated local system files compatible with the processor, a method of providing at least some of the servers in the system with secure, executable, applications related to a service, wherein the applications are executed in a secure environment, wherein the applications each include an object executable by at least some of the different operating systems for performing a task related to the service, the method comprising:	To the extent the preamble is limiting, HPE and/or its customer practices, through the Accused Instrumentalities, in a system having a plurality of servers with operating systems that differ, operating in disparate computing environments, wherein each server includes a processor and an operating system including a kernel a set of associated local system files compatible with the processor, a method of providing at least some of the servers in the system with secure, executable, applications related to a service, wherein the applications are executed in a secure environment, wherein the applications each include an object executable by at least some of the different operating systems for performing a task related to the service, as claimed. For example, HPE Ezmeral Runtime Enterprise runs on individual servers, including HPE Synergy and HPE ProLiant servers, each of which runs an independent operating system, including for example RHEL or SLES running either on bare metal, through an on-premises virtualized infrastructure, through one or more cloud services, or through any other supported deployment. In an exemplary deployment, two or more servers use different operating systems. The servers operate in disparate computing environments, including because each server is a stand-alone computer and/or each server is unrelated to the other servers due to having independent hardware and, in some instances, independent software.

Claim 1	Accused Instrumentalities
	HPE requires that each server includes a processor with one or more cores available to the OS kernel. HPE further requires each server to have a supported operating system (SLES or RHEL/CentOS), which includes a kernel and associated local system files, including for example libraries such as libc/glibc, configuration files, etc. In the infringing system, at least two servers have different operating systems, for example SLES and RHEL/CentOS, or for another example different versions of SLES and/or RHEL/CentOS. In at least some instances, HPE directly owns, operates, controls, and/or benefits from the claimed system and/or method. In other instances, HPE's customer makes and uses the system and/or method either by following HPE's direction and control, including HPE's documentation, or automatically through the ordinary and expected operation of HPE's software, or a combination thereof. *See* claim limitations below. *See also, e.g.*: HPE Ezmeral Runtime Enterprise is an enterprise-grade container orchestration platform that is designed for the containerization of both cloud-native and non-cloud-native monolithic applications with persistent data. It deploys 100% open-source Kubernetes for orchestration, provides a state-of-the-art file system and data fabric for persistent container storage, and provides enterprises with the ability to deploy non-cloud-native AI and Analytics workloads in containers. Enterprises can now easily extend the agility and efficiency benefits of containers to more of their enterprise applications—running on either bare-metal or virtualized infrastructure, on-premises, in multiple clouds, or at the edge. https://www.hpe.com/psnow/doc/a50004264enw.pdf?jumpid=in_pdp-psnow-qs The offering formerly known as the HPE Ezmeral Container Platform is really focused on a lot more than just containers, and it provides businesses with more than just container orchestration software. The name change to HPE Ezmeral Runtime Enterprise reflects the fact that this is not just a solution for container platform orchestration. This platform offers an incredible wealth of capabilities and features you can use to modernize, deploy, monitor, and manage your applications. https://community.hpe.com/t5/hpe-ezmeral-uncut/hpe-ezmeral-container-platform-is-now-hpe-ezmeral-runtime/ba-p/7151720 ▪ **OS agnostic** – With an application and all its necessary files bundled into one unit – minus an operating system – the container can run on different operating systems, hardware, networks, storage systems and security policies. This means that any environment is compatible, so developers don't need to re-write applications for different servers. https://www.hpe.com/us/en/what-is/caas.html

Claim 1	Accused Instrumentalities

With HPE Ezmeral Runtime Enterprise running on HPE Synergy or HPE ProLiant servers, enterprises can extend the agility and efficiency benefits of containers to more of their enterprise applications—running on either bare metal or virtualized infrastructure, either on-premises, in multiple public clouds, or at the edge.

https://www.hpe.com/psnow/doc/a50003599enw

HPE Ezmeral Runtime Enterprise and HPE Ezmeral ML Ops Capabilities Matrix

	HPE Ezmeral Runtime Enterprise Essentials	HPE Ezmeral Runtime Enterprise	HPE Ezmeral MLOps
Operating Systems (OS)	Yes	Yes	Yes
RHEL OS	Yes	Yes	Yes
SLES OS	Yes	Yes	Yes

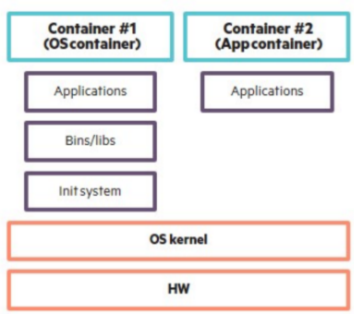
https://www.hpe.com/psnow/doc/a50004264enw.pdf?jumpid=in_pdp-psnow-qs

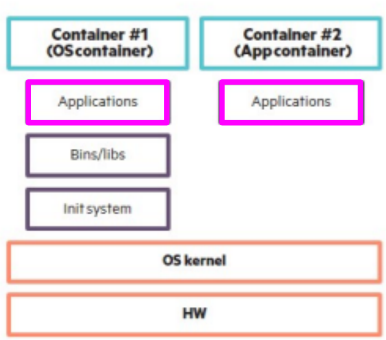
Standard Features

- Leverages portability of containers to run on any infrastructure (HPE or non-HPE) and any public cloud

https://www.hpe.com/psnow/doc/a50004264enw.pdf?jumpid=in_pdp-psnow-qs

Two Linux containers on a single system



Claim 1	Accused Instrumentalities
	https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/whitepaper/pdfs/4AA6-2761ENW.pdf Each license allows the customer to deploy the HPE Ezmeral Container Platform on one Core and 2 terabytes of Storage Capacity. The customer must purchase more licenses if they exceed the allowable amount of Cores or Storage Capacity. As used in this Agreement, Core means a part of a CPU that executes a single stream of compiled instruction code. Each physical processor contains smaller processing units called physical CPU cores. Some processors have two cores, some four, some eight, and so on. Core capacity represents the total number of cores available within a given system. The number of cores is counted as the number of logical cores presented to the product guest OS. For licensing purposes, the number of cores on a given Ezmeral Container Platform host is the number of unique cores available to the kernel in the OS on which the Ezmeral Container Platform software is directly installed, regardless of the number of threads in each core. It equals the product of Core(s) per socket and Socket(s), as shown in the output of https://docs.ezmeral.hpe.com/runtime-enterprise/56/home/about-hpe-ezmeral-container-pl/GEN_End_User_Software_Agreement.html Kernel mode refers to the processor mode that enables software to have full and unrestricted access to the system and its resources. The OS kernel and kernel drivers, such as the file system driver, are loaded into protected memory space and operate in this highly privileged kernel mode. https://www.techtarget.com/searchdatacenter/definition/kernel Instead of using a hypervisor to manage VMs, the figure shows how containers isolate applications into separate environments (containers) that include processor, memory, and networking resources as part of the container itself. This environment provides OS-level virtualization. Containers have their own root; and, users and processes do not perform operations outside of the container environment. The host OS kernel manages container workloads directly, which reduces the overhead involved with managing system resources. This improves efficiency and therefore, improves performance. **Two Linux containers on a single system** 

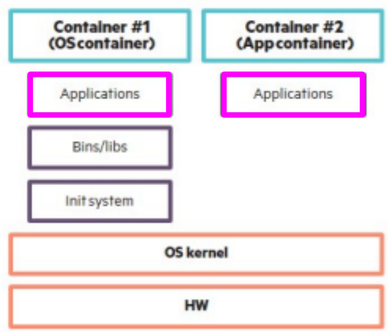
Claim 1	Accused Instrumentalities
	https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/whitepaper/pdfs/4AA6-2761ENW.pdf **Docker container:** A *Docker container* is a lightweight, standalone, executable software package that runs specific services. This software package includes code, runtime, system libraries, configurations, etc. that run as an isolated process in user space. A Docker container is typically used to deploy scalable and repeatable *microservices*. HPE Ezmeral Runtime Enterprise contains innovations around storage, networking, and security to utilize Docker containers as lightweight virtual machines to run Big Data and analytics applications. https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&docLocale=en_US&page=home/about-hpe-ezmeral-container-pl/GEN_Definitions.html **Controller, Gateway, and Worker Hosts** A host is either a physical server or a virtual server, located on your premises or in a public cloud, that is available to HPE Ezmeral Runtime Enterprise. The term **host** and **node** are often used interchangeably. Nodes are hosts that are part of a cluster. You must have a supported operating system installed on hosts before they can be used in HPE Ezmeral Runtime Enterprise. Hosts have different requirements depending on their functions. See Host Requirements. https://docs.ezmeral.hpe.com/runtime-enterprise/56/reference/universal-concepts/Controller_Gateway_and_Worker_Hosts.html

Claim 1	Accused Instrumentalities
	Kubernetes Cluster Nodes A deployment of HPE Ezmeral Runtime Enterprise can include multiple Kubernetes clusters. A host that is part of a Kubernetes cluster is referred to in Kubernetes as a node. Each Kubernetes cluster has its own control plane, consisting of at least one control plane node. The Kubernetes control plane is separate from the Platform Control Plane. A high-availability Kubernetes cluster has multple control plane nodes, as described in High Availability. Kubernetes clusters contain worker nodes that run the containers and pods that process jobs in HPE Ezmeral Runtime Enterprise. For more information about hosts and Kubernetes clusters, see Controller, Gateway, and Worker Hosts. https://docs.ezmeral.hpe.com/runtime-enterprise/56/reference/kubernetes/Kubernetes_Physical_Architecture.html#v52_k8s-kubernetes-physical-architecture__k8s-cluster-architecture
[1a] storing in memory accessible to at least some of the servers a plurality of secure containers of application software, each container comprising one or more of the executable applications and a set of associated system files required to execute the one or more applications, for use with a local kernel residing permanently on one of the servers;	The method practiced by HPE and/or its customer through the Accused Instrumentalities includes a step of storing in memory accessible to at least some of the servers a plurality of secure containers of application software, each container comprising one or more of the executable applications and a set of associated system files required to execute the one or more applications, for use with a local kernel residing permanently on one of the servers. For example, HPE Ezmeral Runtime Enterprise stores application containers, sometimes called Docker containers, container images, Kubernetes containers, or Kubernetes pods, in persistent storage available to each node running the application. The terms "node" and "host" are both used to refer to the claimed server. The container might be in a format defined by the Open Container Initiative. This storage may be physically attached to the server or connected through any supported interconnect, including over a network. In addition to ~~Each container includes~~ the application software ~~as well as~~ , each container includes associated system files, including a Linux user space required to execute the application, for example libc/glibc and other shared libraries, configuration files, etc. necessary for the application. For example, the container includes a base OS image, provided by HPE or by a third

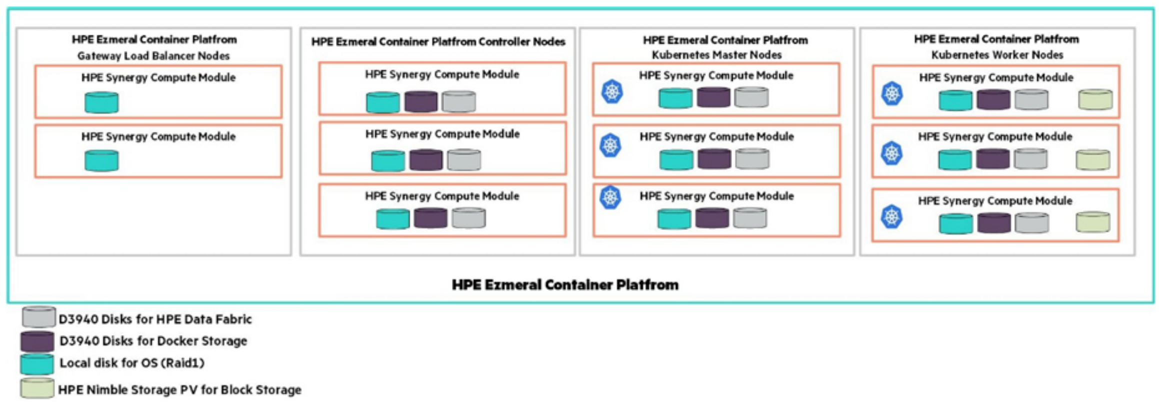
Claim 1	Accused Instrumentalities
	party, such as a CentOS, RHEL, or Ubuntu base image. The container is compatible with the host kernel, for example because the container libraries are linked against the Linux kernel, and the supported host operating systems also use the Linux kernel, which has a stable binary interface. The containers are secure containers as claimed. For example, the data within an individual container is insulated from the effects of other containers except to the extent the container is specifically configured to allow other containers to modify its data, for example using a shared volume. *See, e.g.*: **Pod:** For Kubernetes, a *pod* is a group of containers deployed on a single host. **Data Fabric cluster:** This is a Kubernetes cluster that is used for HPE Ezmeral Data Fabric storage. A Data Fabric cluster is a Custom Resource in Kubernetes that is supported by operators in HPE Ezmeral Runtime Enterprise. **Data Fabric CR:** This typically refers to the Custom Resource specification for a Data Fabric cluster that is supported by an HPE Ezmeral Runtime Enterprise `dataplatform` operator. It specifies each type of pod that the cluster would comprise. The per-pod specification may include CPU, memory, disk, and port requirements. Together with node labels and annotations, the Data Fabric CR influences the placement and scheduling of cluster pods by Kubernetes. HPE Ezmeral Runtime Enterprise creates and applies the Data Fabric CR when creating the first Data Fabric cluster. The Data Fabric CR may be subsequently patched/modified when expanding the cluster, or by a user with suitable privileges. **Docker container:** A *Docker container* is a lightweight, standalone, executable software package that runs specific services. This software package includes code, runtime, system libraries, configurations, etc. that run as an isolated process in user space. A Docker container is typically used to deploy scalable and repeatable *microservices*. HPE Ezmeral Runtime Enterprise contains innovations around storage, networking, and security to utilize Docker containers as lightweight virtual machines to run Big Data and analytics applications. https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&docLocale=en_US&page=home/about-hpe-ezmeral-container-pl/GEN_Definitions.html

Claim 1	Accused Instrumentalities

Two Linux containers on a single system



https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/whitepaper/pdfs/4AA6-2761ENW.pdf



Storage

The HPE Synergy D3940 Storage Module provides solid state disks and hard disk drives to local systems where it is consumed by HPE Ezmeral Data Fabric, and optionally by compute nodes as boot devices. HPE Nimble Storage dynamically provides Persistent Volume (PV) for containers using Dynamic Volume Provisioner which is integrated with the HPE CSI Driver.

https://www.hpe.com/psnow/doc/a50002075enw

HPE Nimble Storage

HPE Nimble Storage AF40 is used to provide persistent, block storage in this solution. The HPE Nimble Storage array for Docker data provides the storage volume to host the repository, to store container images, and also provides persistent volume for applications.

Claim 1	Accused Instrumentalities
	https://www.hpe.com/psnow/doc/a50002075enw Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/ An application container is a stand-alone, all-in-one package for a software application. Containers include the application binaries, plus the software dependencies and the hardware requirements needed to run, all wrapped up into an independent, self-contained unit. https://developer.hpe.com/blog/kubernetes-application-containers-managing-containers-and-cluster-resour/ Because each application container creates an isolated environment for its application, the resources allocated to it are the entire machine. Other copies of the same container are "unaware" of each other. https://developer.hpe.com/blog/kubernetes-application-containers-managing-containers-and-cluster-resour/ • **OS agnostic** – With an application and all its necessary files bundled into one unit – minus an operating system – the container can run on different operating systems, hardware, networks, storage systems and security policies. This means that any environment is compatible, so developers don't need to re-write applications for different servers. https://www.hpe.com/us/en/what-is/caas.html 6. **Do Docker containers package up the entire OS and make it easier to deploy?** Docker containers do not package up the OS. They package up the applications with everything that the application needs to run. The engine is installed on top of the OS running on a host. Containers share the OS kernel allowing a single host to run multiple containers. https://www.docker.com/blog/the-10-most-common-questions-it-admins-ask-about-docker/

Claim 1	Accused Instrumentalities
	Kubernetes namespaces have the following uses: • **Isolation:** Teams, projects, and customers exist in their own environment within a cluster, and do not impact each other's work. https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp55hen_us&docLocale=en_US&page=reference/universal-concepts/Namespaces.html Using containers isolates software and allows it to work independently across different operating systems, hardware, networks, storage systems, and security policies. It allows the container-based application to transition seamlessly through development, testing, and production environments. Because an operating system is not packed into the container, each container uses minimal computing resources, making it light and easy to install. https://www.hpe.com/us/en/what-is/containers.html

Claim 1	Accused Instrumentalities
	# About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. ## Storage drivers versus Docker volumes Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Images and layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

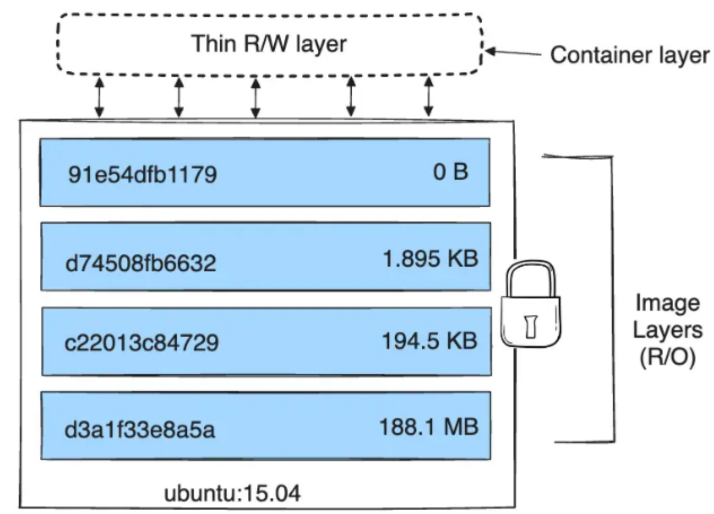
```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

https://docs.docker.com/storage/storagedriver/

</td></tr>
</table>

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  https://docs.docker.com/storage/storagedriver/

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Volumes

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:

https://kubernetes.io/docs/concepts/storage/volumes/

Container environment

The Kubernetes Container environment provides several important resources to Containers:

- A filesystem, which is a combination of an image and one or more volumes.
- Information about the Container itself.
- Information about other objects in the cluster.

https://kubernetes.io/docs/concepts/containers/container-environment/

</td></tr>
</table>

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Images

A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment.

You typically create a container image of your application and push it to a registry before referring to it in a Pod.

https://kubernetes.io/docs/concepts/containers/images/

Volumes

On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested.

https://kubernetes.io/docs/concepts/storage/volumes/

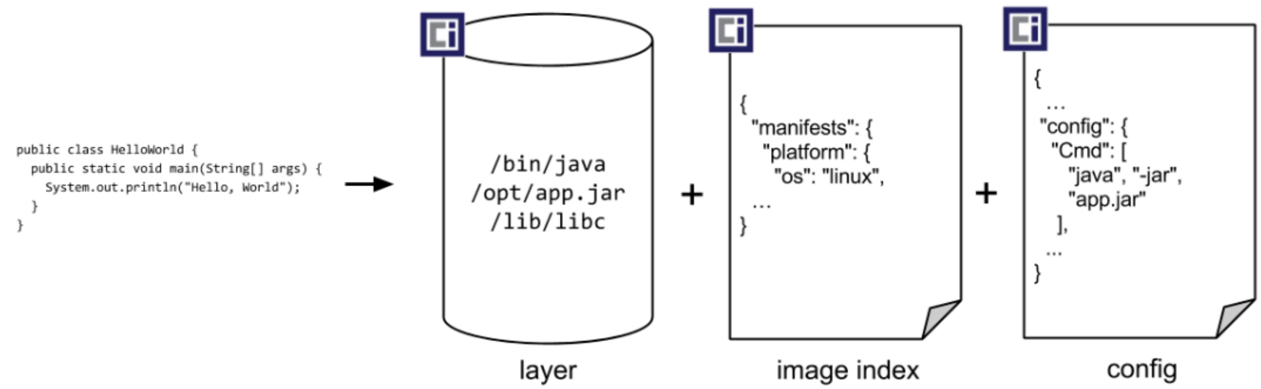
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	Open Container Initiative **Image Format Specification** This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Overview

At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.



https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

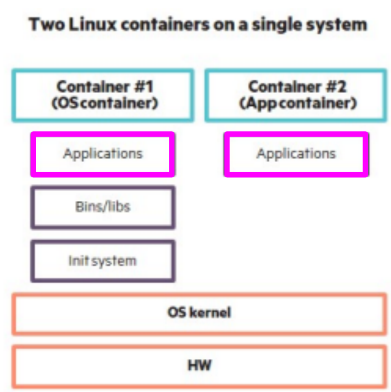
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	# OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 1	Accused Instrumentalities
	Layer • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 1	Accused Instrumentalities
	• **rootfs** *object*, REQUIRED The rootfs key references the layer content addresses used by the image. This makes the image config hash depend on the filesystem hash. ◦ **type** *string*, REQUIRED MUST be set to `layers`. Implementations MUST generate an error if they encounter a unknown value while verifying or unpacking an image. ◦ **diff_ids** *array of strings*, REQUIRED An array of layer content hashes (`DiffIDs`), in order from first to last. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md **Controller, Gateway, and Worker Hosts** A host is either a physical server or a virtual server, located on your premises or in a public cloud, that is available to HPE Ezmeral Runtime Enterprise. The term **host** and **node** are often used interchangeably. Nodes are hosts that are part of a cluster. You must have a supported operating system installed on hosts before they can be used in HPE Ezmeral Runtime Enterprise. Hosts have different requirements depending on their functions. See Host Requirements. https://docs.ezmeral.hpe.com/runtime-enterprise/56/reference/universal-concepts/Controller_Gateway_and_Worker_Hosts.html

Claim 1	Accused Instrumentalities
	Kubernetes Cluster Nodes A deployment of HPE Ezmeral Runtime Enterprise can include multiple Kubernetes clusters. A host that is part of a Kubernetes cluster is referred to in Kubernetes as a node. Each Kubernetes cluster has its own control plane, consisting of at least one control plane node. The Kubernetes control plane is separate from the Platform Control Plane. A high-availability Kubernetes cluster has multple control plane nodes, as described in High Availability. Kubernetes clusters contain worker nodes that run the containers and pods that process jobs in HPE Ezmeral Runtime Enterprise. For more information about hosts and Kubernetes clusters, see Controller, Gateway, and Worker Hosts. https://docs.ezmeral.hpe.com/runtime-enterprise/56/reference/kubernetes/Kubernetes_Physical_Architecture.html#v52_k8s-kubernetes-physical-architecture__k8s-cluster-architecture

Claim 1	Accused Instrumentalities
[1b] wherein the set of associated system files are compatible with a local kernel of at least some of the plurality of different operating systems,	In the method practiced by HPE and/or its customer through the Accused Instrumentalities, the set of associated system files are compatible with a local kernel of at least some of the plurality of different operating systems. The associated system files in the container are compatible with the host kernel, for example because they are linked against the Linux kernel and the supported host operating systems also use the Linux kernel, which has a stable binary interface. *See* discussion in element [1a] above. *See, e.g.*: **Docker container:** A *Docker container* is a lightweight, standalone, executable software package that runs specific services. This software package includes code, runtime, system libraries, configurations, etc. that run as an isolated process in user space. A Docker container is typically used to deploy scalable and repeatable *microservices*. HPE Ezmeral Runtime Enterprise contains innovations around storage, networking, and security to utilize Docker containers as lightweight virtual machines to run Big Data and analytics applications. https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&docLocale=en_US&page=home/about-hpe-ezmeral-container-pl/GEN_Definitions.html - **OS agnostic** – With an application and all its necessary files bundled into one unit – minus an operating system – the container can run on different operating systems, hardware, networks, storage systems and security policies. This means that any environment is compatible, so developers don't need to re-write applications for different servers. https://www.hpe.com/us/en/what-is/caas.html 

Claim 1	Accused Instrumentalities
	https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/whitepaper/pdfs/4AA6-2761ENW.pdf

Claim 1	Accused Instrumentalities
[1c] the containers of application software excluding a kernel,	In the method practiced by HPE and/or its customer through the Accused Instrumentalities, the containers of application software exclude a kernel. *See, e.g.*: **Docker container:** A *Docker container* is a lightweight, standalone, executable software package that runs specific services. This software package includes code, runtime, system libraries, configurations, etc. that run as an isolated process in user space. A Docker container is typically used to deploy scalable and repeatable *microservices*. HPE Ezmeral Runtime Enterprise contains innovations around storage, networking, and security to utilize Docker containers as lightweight virtual machines to run Big Data and analytics applications. https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&docLocale=en_US&page=home/about-hpe-ezmeral-container-pl/GEN_Definitions.html 6. **Do Docker containers package up the entire OS and make it easier to deploy?** Docker containers do not package up the OS. They package up the applications with everything that the application needs to run. The engine is installed on top of the OS running on a host. Containers share the OS kernel allowing a single host to run multiple containers. https://www.docker.com/blog/the-10-most-common-questions-it-admins-ask-about-docker/ Containers and VMs perform somewhat similar functions in that they provide virtualized environments in which software applications can run separately from the rest of the system. But these technologies are very different and are used in different situations. Each virtual machine runs both an OS and the application, while containers share a single OS via a kernel, making them more lightweight and portable. https://www.hpe.com/us/en/what-is/containers.html

Claim 1	Accused Instrumentalities
[1d] wherein some or all of the associated system files within a container stored in memory are utilized in place of the associated local system files that remain resident on the server,	In the method practiced by HPE and/or its customer through the Accused Instrumentalities, some or all of the associated system files within a container stored in memory are utilized in place of the associated local system files that remain resident on the server. For example, each container will utilize its own associated ~~local~~ system files, including libraries such as libc/glibc and configuration files, not the corresponding associated local system files (*e.g.*, libraries and configuration files of the host OS). As described above and below, in the Accused Instrumentalities the associated system files provide at least some of the same functionalities as the associated local system files. The host/node's associated local system files remain resident on the host/node, for example for use by system processes or applications outside the container environment. *See, e.g.*: **Docker container:** A *Docker container* is a lightweight, standalone, executable software package that runs specific services. This software package includes code, runtime, system libraries, configurations, etc. that run as an isolated process in user space. A Docker container is typically used to deploy scalable and repeatable *microservices*. HPE Ezmeral Runtime Enterprise contains innovations around storage, networking, and security to utilize Docker containers as lightweight virtual machines to run Big Data and analytics applications. https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&docLocale=en_US&page=home/about-hpe-ezmeral-container-pl/GEN_Definitions.html An application container is a stand-alone, all-in-one package for a software application. Containers include the application binaries, plus the software dependencies and the hardware requirements needed to run, all wrapped up into an independent, self-contained unit. https://developer.hpe.com/blog/kubernetes-application-containers-managing-containers-and-cluster-resour/ - **OS agnostic** – With an application and all its necessary files bundled into one unit – minus an operating system – the container can run on different operating systems, hardware, networks, storage systems and security policies. This means that any environment is compatible, so developers don't need to re-write applications for different servers. https://www.hpe.com/us/en/what-is/caas.html

Claim 1	Accused Instrumentalities
[1e] wherein said associated system files utilized in place of the associated local system files are copies or modified copies of the associated local system files that remain resident on the server,	In the method practiced by HPE and/or its customer through the Accused Instrumentalities, said associated system files utilized in place of the associated local system files are copies or modified copies of the associated local system files that remain resident on the server. For example, in some cases the host OS and container will use one or more identical system files, for example when both the host and the container incorporate the same Linux distribution version, or when both host and container use the same version of libc. In the case where the associated system files are identical to the associated local system files, they are copies thereof. In other cases modified copies are used instead, for example when different versions of the same library, or configuration files with different parameters, are used by the host and container. *See, e.g.*: **Docker container:** A *Docker container* is a lightweight, standalone, executable software package that runs specific services. This software package includes code, runtime, system libraries, configurations, etc. that run as an isolated process in user space. A Docker container is typically used to deploy scalable and repeatable *microservices*. HPE Ezmeral Runtime Enterprise contains innovations around storage, networking, and security to utilize Docker containers as lightweight virtual machines to run Big Data and analytics applications. https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&docLocale=en_US&page=home/about-hpe-ezmeral-container-pl/GEN_Definitions.html `COPY` and `ADD`: These commands copy files and directories from your local filesystem into the Docker image. They are often used to include your application code, configuration files, and dependencies. https://medium.com/@swalperen3008/what-is-dockerize-and-dockerize-your-project-a-step-by-step-guide-899c48a34df6 **Container images** A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/

Claim 1	Accused Instrumentalities
[1f] and wherein the application software cannot be shared between the plurality of secure containers of application software,	In the method practiced by HPE and/or its customer through the Accused Instrumentalities, the application software cannot be shared between the plurality of secure containers of application software. For example, each container has an isolated runtime environment that cannot be accessed by other containers, for example including a per-container writeable layer or other ephemeral per-container storage. For another example, when the plurality of secure containers each corresponds to a different container image, each container cannot access another container's image and therefore application software. *See, e.g.*: **Docker container:** A *Docker container* is a lightweight, standalone, executable software package that runs specific services. This software package includes code, runtime, system libraries, configurations, etc. that run as an isolated process in user space. A Docker container is typically used to deploy scalable and repeatable *microservices*. HPE Ezmeral Runtime Enterprise contains innovations around storage, networking, and security to utilize Docker containers as lightweight virtual machines to run Big Data and analytics applications. https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&docLocale=en_US&page=home/about-hpe-ezmeral-container-pl/GEN_Definitions.html Kubernetes namespaces have the following uses: • **Isolation:** Teams, projects, and customers exist in their own environment within a cluster, and do not impact each other's work. https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp55hen_us&docLocale=en_US&page=reference/universal-concepts/Namespaces.html Because each application container creates an isolated environment for its application, the resources allocated to it are the entire machine. Other copies of the same container are "unaware" of each other. https://developer.hpe.com/blog/kubernetes-application-containers-managing-containers-and-cluster-resour/

Claim 1	Accused Instrumentalities
	# About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. ## Storage drivers versus Docker volumes Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Images and layers

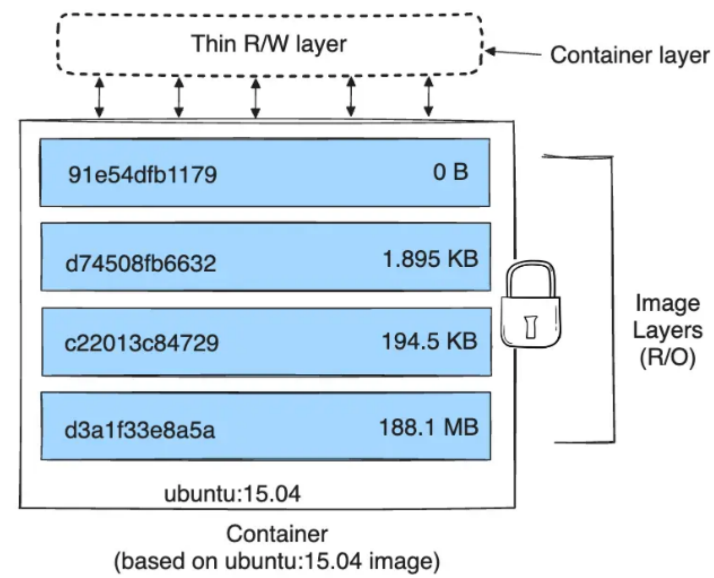
A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

https://docs.docker.com/storage/storagedriver/
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
[1g] and wherein each of the containers has a unique root file system that is different from an operating system's root file system.	In the method practiced by HPE and/or its customer through the Accused Instrumentalities, each of the containers has a unique root file system that is different from an operating system's root file system. For example, the container's root file system comprises the image layer(s), an ephemeral writeable layer (e.g., in Docker terminology the container layer), and optionally one or more volumes. This root file system is distinct and isolated from the host operating system's root file system. *See, e.g.*: Using containers isolates software and allows it to work independently across different operating systems, hardware, networks, storage systems, and security policies. It allows the container-based application to transition seamlessly through development, testing, and production environments. Because an operating system is not packed into the container, each container uses minimal computing resources, making it light and easy to install. https://www.hpe.com/us/en/what-is/containers.html **Node storage:** *Node storage* is storage space available for backing the root file systems of containers. Each HPE Ezmeral Runtime Enterprise host contributes node storage space that is used by the virtual nodes (Docker containers) assigned to that host. The Platform Administrator may optionally specify a quota limiting how much node storage a tenant's virtual nodes may consume. https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&docLocale=en_US&page=home/about-hpe-ezmeral-container-pl/GEN_Definitions.html

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

About storage drivers

To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way.

Storage drivers versus Docker volumes

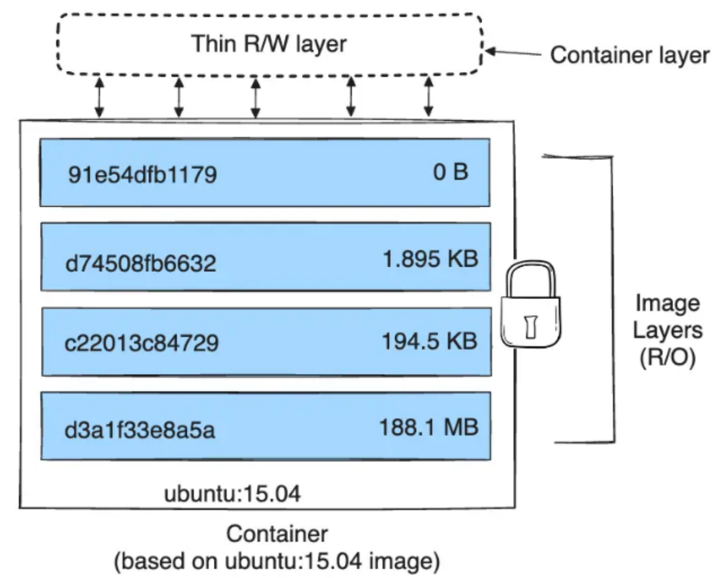
Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer.

Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance.

https://docs.docker.com/storage/storagedriver/
</td></tr>
</table>

Claim 1	Accused Instrumentalities

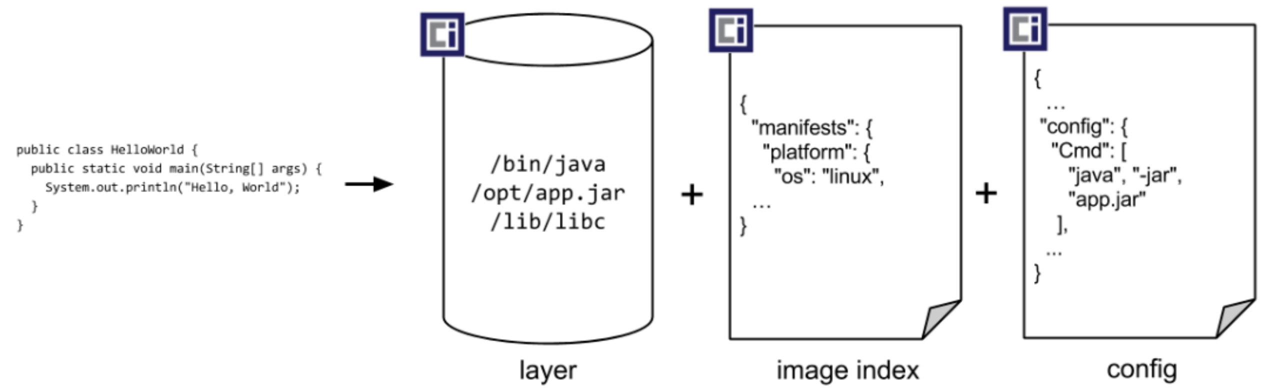
	Images and layers
	
	A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:
	
	`# syntax=docker/dockerfile:1`
	
	`FROM ubuntu:22.04`
	`LABEL org.opencontainers.image.authors="org@example.com"`
	`COPY . /app`
	`RUN make /app`
	`RUN rm -r $HOME/.cache`
	`CMD python /app/app.py`
	
	This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.
	
	https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	Volumes Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/ **Container environment** The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

Claim 1	Accused Instrumentalities
	Images A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/ **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	Open Container Initiative **Image Format Specification** This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 1	Accused Instrumentalities
	Overview At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.  https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

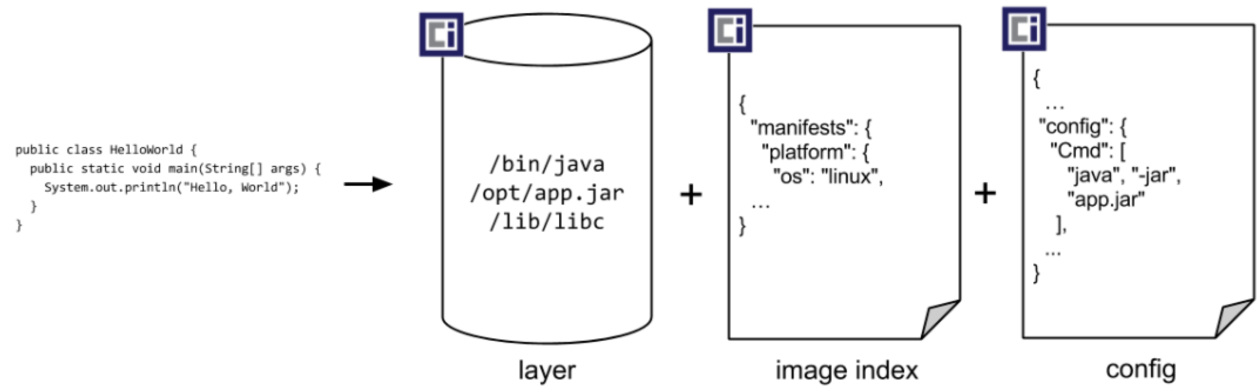
Claim 1	Accused Instrumentalities
	OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 1	Accused Instrumentalities
	Layer • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 1	Accused Instrumentalities
	• **rootfs** *object*, REQUIRED The rootfs key references the layer content addresses used by the image. This makes the image config hash depend on the filesystem hash. ○ **type** *string*, REQUIRED MUST be set to `layers`. Implementations MUST generate an error if they encounter a unknown value while verifying or unpacking an image. ○ **diff_ids** *array of strings*, REQUIRED An array of layer content hashes (`DiffIDs`), in order from first to last. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 2

Claim 2	Accused Instrumentalities
2. A method as defined in claim 1, wherein each container has an execution file associated therewith for starting the one or more applications.	HPE and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 1, wherein each container has an execution file associated therewith for starting the one or more applications. For example, a container image has an associated image configuration comprising information for starting the one or more applications. This can be an Open Containers Initiative image configuration. *See, e.g.*: **Open Container Initiative** **Image Format Specification** This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 2	Accused Instrumentalities
	## Overview At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.  https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 2	Accused Instrumentalities
	# OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 2	Accused Instrumentalities
	• **config** *object*, OPTIONAL The execution parameters which SHOULD be used as a base when running a container using the image. This field can be `null`, in which case any execution parameters should be specified at creation of the container. ○ **Env** *array of strings*, OPTIONAL Entries are in the format of `VARNAME=VARVALUE`. These values act as defaults and are merged with any specified when creating a container. ○ **Entrypoint** *array of strings*, OPTIONAL A list of arguments to use as the command to execute when the container starts. These values act as defaults and may be replaced by an entrypoint specified when creating a container. ○ **Cmd** *array of strings*, OPTIONAL Default arguments to the entrypoint of the container. These values act as defaults and may be replaced by any specified when creating a container. If an `Entrypoint` value is not specified, then the first entry of the `Cmd` array SHOULD be interpreted as the executable to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 6

Claim 6	Accused Instrumentalities
6. A method as defined in claim 2, comprising the step of assigning a unique associated identity to each of a plurality of the containers, wherein the identity includes at least one of IP address, host name, and MAC address.	HPE and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 2, comprising the step of assigning a unique associated identity to each of a plurality of the containers, wherein the identity includes at least one of IP address, host name, and MAC address. For example, Kubernetes containers have an associated hostname, which in the case of a single-container Pod is the unique identity of that container. For another example, Kubernetes pods have an associated hostname, which is unique. For another example, a networked Kubernetes pod has an assigned IPv4 and/or IPv6 address. For another example, a Docker container has an IP address and a hostname. *See, e.g.*: **Container information** The *hostname* of a Container is the name of the Pod in which the Container is running. It is available through the `hostname` command or the `gethostname` function call in libc. The Pod name and namespace are available as environment variables through the downward API. User defined environment variables from the Pod definition are also available to the Container, as are any environment variables specified statically in the container image. https://kubernetes.io/docs/concepts/containers/container-environment/

Claim 6	Accused Instrumentalities
	IP address and hostname By default, the container gets an IP address for every Docker network it attaches to. A container receives an IP address out of the IP subnet of the network. The Docker daemon performs dynamic subnetting and IP address allocation for containers. Each network also has a default subnet mask and gateway. You can connect a running container to multiple networks, either by passing the `--network` flag multiple times when creating the container, or using the `docker network connect` command for already running containers. In both cases, you can use the `--ip` or `--ip6` flags to specify the container's IP address on that particular network. In the same way, a container's hostname defaults to be the container's ID in Docker. You can override the hostname using `--hostname`. When connecting to an existing network using `docker network connect`, you can use the `--alias` flag to specify an additional network alias for the container on that network. https://docs.docker.com/network/

Claim 9

<table>
<tr><th>Claim 9</th><th>Accused Instrumentalities</th></tr>
<tr><td>9. A method as defined in claim 2, wherein server information related to hardware resource usage including at least one of CPU memory, network bandwidth, and disk allocation is associated with at least some of the containers prior to the applications within the containers being executed.</td><td>HPE and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 2, wherein server information related to hardware resource usage including at least one of CPU memory, network bandwidth, and disk allocation is associated with at least some of the containers prior to the applications within the containers being executed.

For example, Kubernetes tracks and limits resource usage, including CPU and memory resources. For another example, Docker tracks and limits resource usage, including CPU and memory resources.

See, e.g.:

<u>Resource Management for Pods and Containers</u>

When you specify a <u>Pod</u>, you can optionally specify how much of each resource a <u>container</u> needs. The most common resources to specify are CPU and memory (RAM); there are others.

When you specify the resource request for containers in a Pod, the <u>kube-scheduler</u> uses this information to decide which node to place the Pod on. When you specify a resource limit for a container, the <u>kubelet</u> enforces those limits so that the running container is not allowed to use more of that resource than the limit you set. The kubelet also reserves at least the request amount of that system resource specifically for that container to use.

Requests and limits
If the node where a Pod is running has enough of a resource available, it's possible (and allowed) for a container to use more resource than its <code>request</code> for that resource specifies. However, a container is not allowed to use more than its resource <code>limit</code>.</td></tr>
</table>

Claim 9	Accused Instrumentalities
	For example, if you set a `memory` request of 256 MiB for a container, and that container is in a Pod scheduled to a Node with 8GiB of memory and no other Pods, then the container can try to use more RAM. If you set a `memory` limit of 4GiB for that container, the kubelet (and container runtime) enforce the limit. The runtime prevents the container from using more than the configured resource limit. For example: when a process in the container tries to consume more than the allowed amount of memory, the system kernel terminates the process that attempted the allocation, with an out of memory (OOM) error. Limits can be implemented either reactively (the system intervenes once it sees a violation) or by enforcement (the system prevents the container from ever exceeding the limit). Different runtimes can have different ways to implement the same restrictions. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ **Runtime options with Memory, CPUs, and GPUs** By default, a container has no resource constraints and can use as much of a given resource as the host's kernel scheduler allows. Docker provides ways to control how much memory, or CPU a container can use, setting runtime configuration flags of the `docker run` command. This section provides details on when you should set such limits and the possible implications of setting them. **Limit a container's access to memory** Docker can enforce hard or soft memory limits. • Hard limits lets the container use no more than a fixed amount of memory.

Claim 9	Accused Instrumentalities
	• Soft limits lets the container use as much memory as it needs unless certain conditions are met, such as when the kernel detects low memory or contention on the host machine. https://docs.docker.com/config/containers/resource_constraints/

Claim 10

Claim 10	Accused Instrumentalities
10. A method as defined in claim 2, wherein in operation when an application residing within a container is executed, said application has no access to system files or applications in other containers or to system files within the operating system during execution thereof.	HPE and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 2, wherein in operation when an application residing within a container is executed, said application has no access to system files or applications in other containers or to system files within the operating system during execution thereof. *See, e.g.*: **Docker container:** A *Docker container* is a lightweight, standalone, executable software package that runs specific services. This software package includes code, runtime, system libraries, configurations, etc. that run as an isolated process in user space. A Docker container is typically used to deploy scalable and repeatable *microservices*. HPE Ezmeral Runtime Enterprise contains innovations around storage, networking, and security to utilize Docker containers as lightweight virtual machines to run Big Data and analytics applications. https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&docLocale=en_US&page=home/about-hpe-ezmeral-container-pl/GEN_Definitions.html Kubernetes namespaces have the following uses: • **Isolation:** Teams, projects, and customers exist in their own environment within a cluster, and do not impact each other's work. https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp55hen_us&docLocale=en_US&page=reference/universal-concepts/Namespaces.html Because each application container creates an isolated environment for its application, the resources allocated to it are the entire machine. Other copies of the same container are "unaware" of each other. https://developer.hpe.com/blog/kubernetes-application-containers-managing-containers-and-cluster-resour/

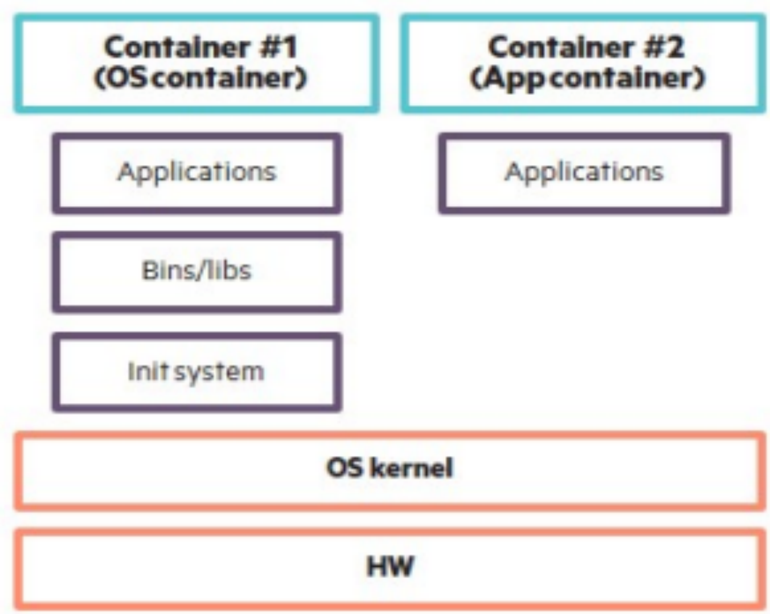
U.S. Patent No. 7,784,058 vs. HPE

Accused Instrumentalities: HPE products and services using user mode critical system elements as shared libraries, including without limitation HPE's Ezmeral Runtime Enterprise (including without limitation both Ezmeral Runtime Enterprise and Ezmeral Runtime Enterprise Essentials, in each case including when marketed, sold, and/or licensed as part of or associated with HPE's GreenLake branding, e.g. "HPE GreenLake for containers" which "is built on HPE Ezmeral Container Platform") ~~and HPE GreenLake~~, and all versions and variations thereof since the issuance of the asserted patent.

Each Accused Instrumentality infringes the claims in substantially the same way, and the evidence shown in this chart is similarly applicable to each Accused Instrumentality. Each claim limitation is literally infringed by each Accused Instrumentality. However, to the extent any claim limitation is not met literally, it is nonetheless met under the doctrine of equivalents because the differences between the claim limitation and each Accused Instrumentality would be insubstantial, and each Accused Instrumentality performs substantially the same function, in substantially the same way, to achieve the same result as the claimed invention. Notably, Defendant has not yet articulated which, if any, particular claim limitations it believes are not met by the Accused Instrumentalities.

Claim 1

Claim 1	Accused Instrumentalities
[1pre] 1. A computing system for executing a plurality of software applications comprising:	To the extent the preamble is limiting, each Accused Instrumentality comprises or constitutes a computing system for executing a plurality of software applications as claimed. *See* claim limitations below. *See also, e.g.*: **HPE Ezmeral Runtime Enterprise** is a unified platform built on open-source Kubernetes and designed for both cloud-native applications and non-cloud-native applications running on any infrastructure; whether on-premises, in multiple public clouds, in a hybrid model, or at the edge. https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&page=home/about-hpe-ezmeral-container-pl/Welcome.html

Claim 1	Accused Instrumentalities
	Containers provide the core runtime abstraction for the user applications. These containers provide isolation between user applications and the rest of the infrastructure. The containers are based on Docker. https://support.hpe.com/hpesc/public/docDisplay?docId=a00097165en_us&docLocale=en_US&page=GUID-6B6676DB-AF5F-4555-B6AB-D2C11A89F320.html Two Linux containers on a single system  https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/whitepaper/pdfs/4AA6-2761ENW.pdf
[1a] a) a processor;	Each Accused Instrumentality comprises a processor. For example, each node/host contains at least one CPU. *See, e.g.*:

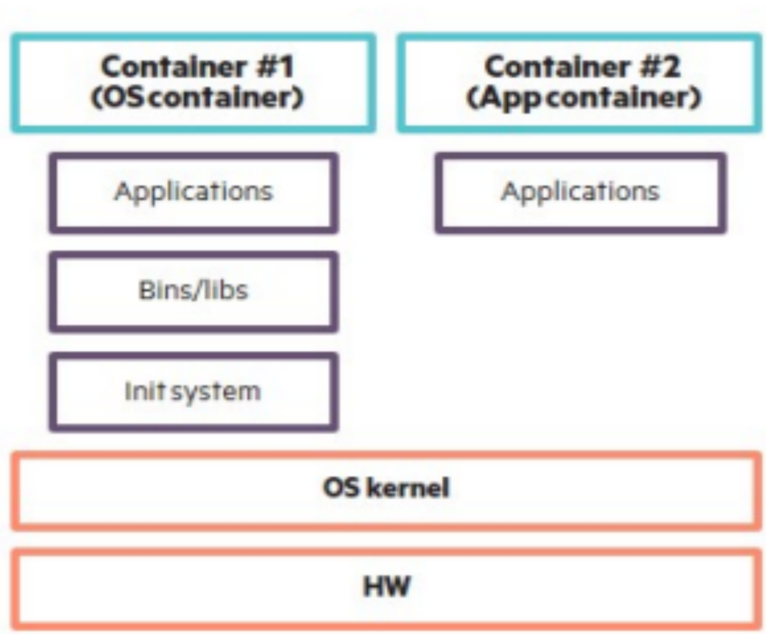
Claim 1	Accused Instrumentalities
	Each license allows the customer to deploy the HPE Ezmeral Container Platform on one Core and 2 terabytes of Storage Capacity. The customer must purchase more licenses if they exceed the allowable amount of Cores or Storage Capacity. As used in this Agreement, Core means a part of a CPU that executes a single stream of compiled instruction code. Each physical processor contains smaller processing units called physical CPU cores. Some processors have two cores, some https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&docLocale=en_US&page=home/about-hpe-ezmeral-container-pl/GEN_End_User_Software_Agreement.html
[1b] b) an operating system having an operating system kernel having OS critical system elements (OSCSEs) for running in kernel mode using said processor; and,	Each Accused Instrumentality comprises an operating system having an operating system kernel having OS critical system elements (OSCSEs) for running in kernel mode using said processor. For example, the OSCSEs include kernel-mode functions similar to the functionalities provided by user-space libraries such as glibc. These are implemented in kernel-space to handle tasks such as (without limitation) memory management (kmalloc(), kfree(), etc,) at kernel level. *See, e.g.*: Containers provide the core runtime abstraction for the user applications. These containers provide isolation between user applications and the rest of the infrastructure. The containers are based on Docker. https://support.hpe.com/hpesc/public/docDisplay?docId=a00097165en_us&docLocale=en_US&page=GUID-6B6676DB-AF5F-4555-B6AB-D2C11A89F320.html Each license allows the customer to deploy the HPE Ezmeral Container Platform on one Core and 2 terabytes of Storage Capacity. The customer must purchase more licenses if they exceed the allowable amount of Cores or Storage Capacity. As used in this Agreement, Core means a part of a CPU that executes a single stream of compiled instruction code. Each physical processor contains smaller processing units called physical CPU cores. Some processors have two cores, some https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&docLocale=en_US&page=home/about-hpe-ezmeral-container-pl/GEN_End_User_Software_Agreement.html

Claim 1	Accused Instrumentalities
	HPE Ezmeral Runtime Enterprise supports the following operating systems:

HPE Ezmeral Runtime Enterprise Version	CentOS Support	RHEL Support	SUSE Support

https://support.hpe.com/hpesc/public/docDisplay?docId=a00ecp54hen_us&page=home/about-hpe-ezmeral-container-pl/GEN_OS_Support.html

Two Linux containers on a single system



https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/whitepaper/pdfs/4AA6-2761ENW.pdf

Claim 1	Accused Instrumentalities
	Kernel mode Kernel mode refers to the processor mode that enables software to have full and unrestricted access to the system and its resources. The OS kernel and kernel drivers, such as the file system driver, are loaded into protected memory space and operate in this highly privileged kernel mode. https://www.techtarget.com/searchdatacenter/definition/kernel The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system. https://en.wikipedia.org/wiki/Glibc
[1c] c) a shared library having shared library critical system elements (SLCSEs) stored therein for use by the plurality of software applications in user mode and	Each Accused Instrumentality comprises a shared library having shared library critical system elements (SLCSEs) stored therein for use by the plurality of software applications in user mode. For example, the shared library with SLCSEs include the runtime environment, system tools, and dependencies, such as the glibc library and other libraries that replicate OSCSEs, included in the container image (including without limitation in a base image that is included within the container image). *See, e.g.*: Containers provide the core runtime abstraction for the user applications. These containers provide isolation between user applications and the rest of the infrastructure. The containers are based on Docker. https://support.hpe.com/hpesc/public/docDisplay?docId=a00097165en_us&docLocale=en_US&page=GUID-6B6676DB-AF5F-4555-B6AB-D2C11A89F320.html The container starts with a base image, and the microservice is packaged into a container image and then deployed through the container platform. The container platform is based on https://www.hpe.com/us/en/what-is/container-platform.html

Claim 1	Accused Instrumentalities
	Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/ The idea of containerization is to isolate and package the application with all the dependencies in a container. https://community.hpe.com/t5/hpe-blog-uk-ireland-middle-east/containerization-the-next-generation-of-virtualization/ba-p/7154442 Container image files are complete, static and executable versions of an application or service and differ from one technology to another. Docker images are made up of multiple layers, which start with a base image that includes all of the dependencies needed to execute code in a container. Each image has a readable/writable layer on top of static unchanging layers. Because each container has its own specific container layer that customizes that specific container, underlying image layers can be saved and reused in multiple containers. An Open Container Initiative (OCI) https://www.techtarget.com/searchitoperations/definition/container-containerization-or-container-based-virtualization

Claim 1	Accused Instrumentalities
	# About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. ## Storage drivers versus Docker volumes Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities

Images and layers

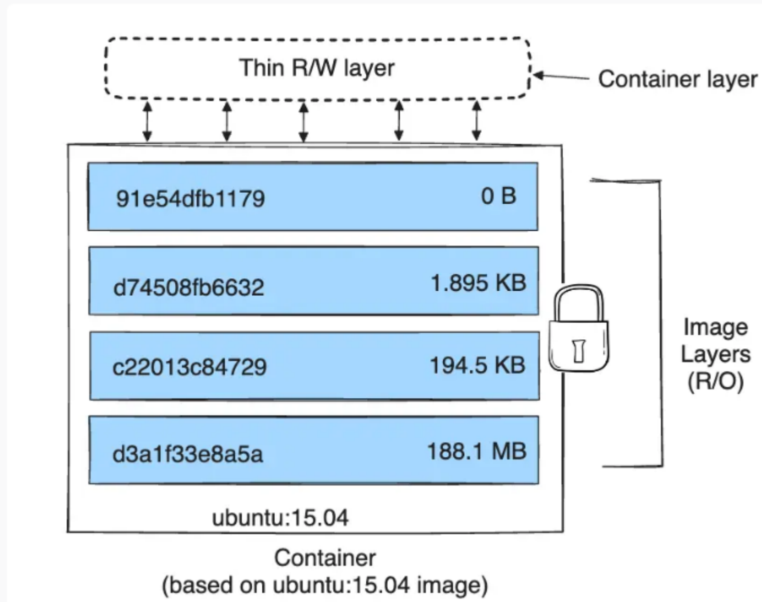
A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

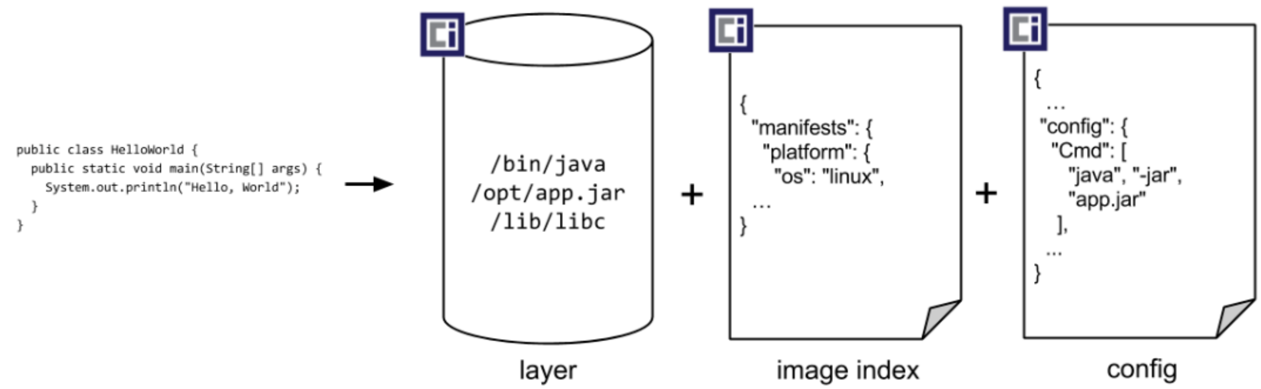
https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	Volumes Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/ **Container environment** The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

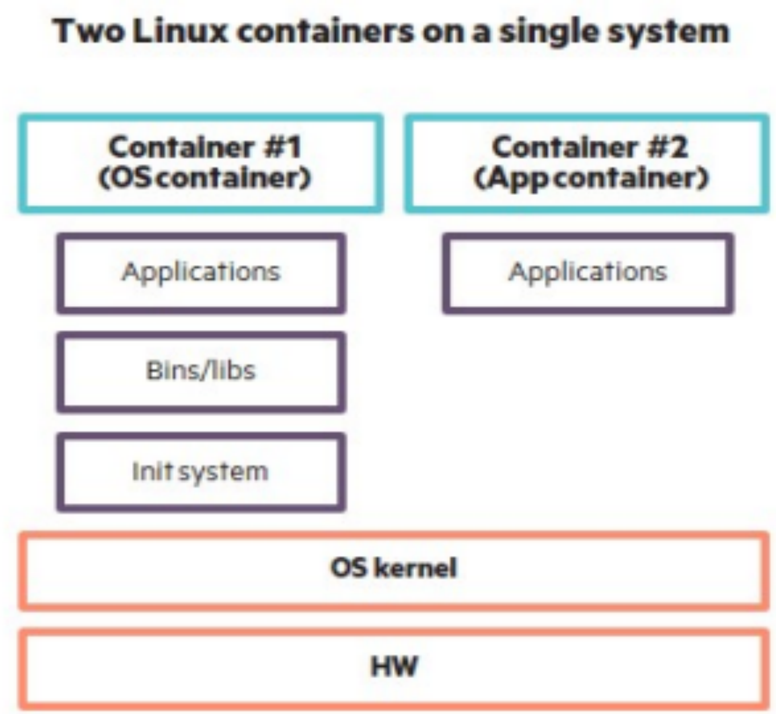
Claim 1	Accused Instrumentalities
	Images A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/ **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	Open Container Initiative **Image Format Specification** This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 1	Accused Instrumentalities
	Overview At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.  https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 1	Accused Instrumentalities
	# OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 1	Accused Instrumentalities
	Layer • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/whitepaper/pdfs/4AA6-2761ENW.pdf

The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system.

https://en.wikipedia.org/wiki/Glibc</td></tr>
<tr><td>[1d] i) wherein some of the SLCSEs stored in the shared library are functional replicas of OSCSEs and are accessible to some of the plurality of software applications and when one of the</td><td>In each Accused Instrumentality, some of the SLCSEs stored in the shared library are functional replicas of OSCSEs and are accessible to some of the plurality of software applications and when one of the SLCSEs is accessed by one or more of the plurality of software applications it forms a part of the one or more of the plurality of software applications.</td></tr>
</table>

Claim 1	Accused Instrumentalities
SLCSEs is accessed by one or more of the plurality of software applications it forms a part of the one or more of the plurality of software applications,	For example, a base image serves as a self-contained unit that encompasses all the necessary components for an application to run, including the application code, runtime environment, system tools, and dependencies (i.e., SLCSEs). The images are based on existing Linux distributions, such as Debian and Ubuntu, including essential system elements (i.e., functional replicas of OSCSEs). Each container image is based on a specific base image, which contains the application code, and dependencies, including system libraries or shared library critical system elements (SLCSEs). The base image forms a part of the container image according to the "layer" model described in the documentation below. When the container runs the image, it creates a runtime instance of that container image. In turn, when one or more applications executes within the container runtime environment, it dynamically links to the SLCSEs stored in the runtime environment, which thereby become a part of the application(s). *See, e.g.*: Hewlett Packard Enterprise provides publicly available base OS images for use in containerized clusters. These images extend the base OS images available from Docker hub by adding several packages that permit HPE Ezmeral Runtime Enterprise to manage container orchestration seamlessly and to improve the security of the container. https://docs.ezmeral.hpe.com/runtime-enterprise/55/app-workbench-5-1/custom-base-images/AWB51_About_Custom_Base_Images.html The idea of containerization is to isolate and package the application with all the dependencies in a container. https://community.hpe.com/t5/hpe-blog-uk-ireland-middle-east/containerization-the-next-generation-of-virtualization/ba-p/7154442 **Container images** A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/

Claim 1	Accused Instrumentalities
	ubuntu Updated 15 days ago Ubuntu is a Debian-based Linux operating system based on free software. Linux IBM Z 386 riscv64 x86-64 ARM ARM 64 PowerPC 64 LE ⬇1B+ · ☆10K+ **debian** Updated 35 minutes ago Debian is a Linux distribution that's composed entirely of free and open-source software. Linux riscv64 x86-64 ARM ARM 64 386 mips64le PowerPC 64 LE IBM Z ⬇1B+ · ☆4.9K https://hub.docker.com/search?image_filter=official&type=image&q=

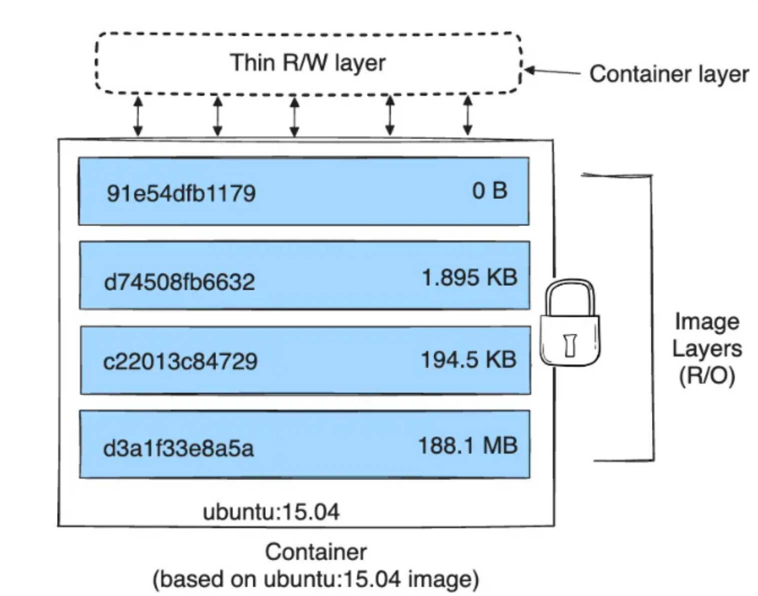
Platform	x86_64 / amd64	arm64 / aarch64	arm (32-bit)	ppc64le	s390x
CentOS	✅	✅		✅	
Debian	✅	✅	✅	✅	
Fedora	✅	✅		✅	
Raspberry Pi OS (32-bit)			✅		
RHEL (s390x)					✅
SLES					✅
Ubuntu	✅	✅	✅	✅	✅
Binaries	✅	✅	✅		

Claim 1	Accused Instrumentalities
	https://docs.docker.com/engine/install/ Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. https://www.techtarget.com/searchitoperations/definition/Docker-image

Claim 1	Accused Instrumentalities
	# About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. ## Storage drivers versus Docker volumes Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities

	Images and layers
	
	A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:
	
	`# syntax=docker/dockerfile:1`
	
	`FROM ubuntu:22.04`
	`LABEL org.opencontainers.image.authors="org@example.com"`
	`COPY . /app`
	`RUN make /app`
	`RUN rm -r $HOME/.cache`
	`CMD python /app/app.py`
	
	This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.
	
	https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  https://docs.docker.com/storage/storagedriver/

Claim 1	Accused Instrumentalities
	# Volumes Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/ ## Container environment The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

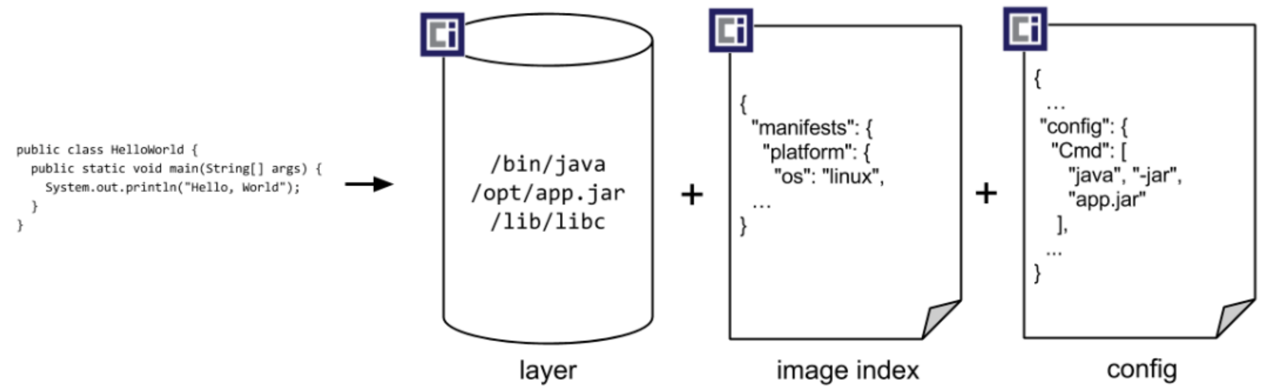
Claim 1	Accused Instrumentalities
	# Images A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/ # Volumes On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	# Open Container Initiative ## Image Format Specification This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Overview

At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.



https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

</td></tr>
</table>

Claim 1	Accused Instrumentalities
	# OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Layer

- Image filesystems are composed of *layers*.
- Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer.
- Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer.
- Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem.

Image JSON

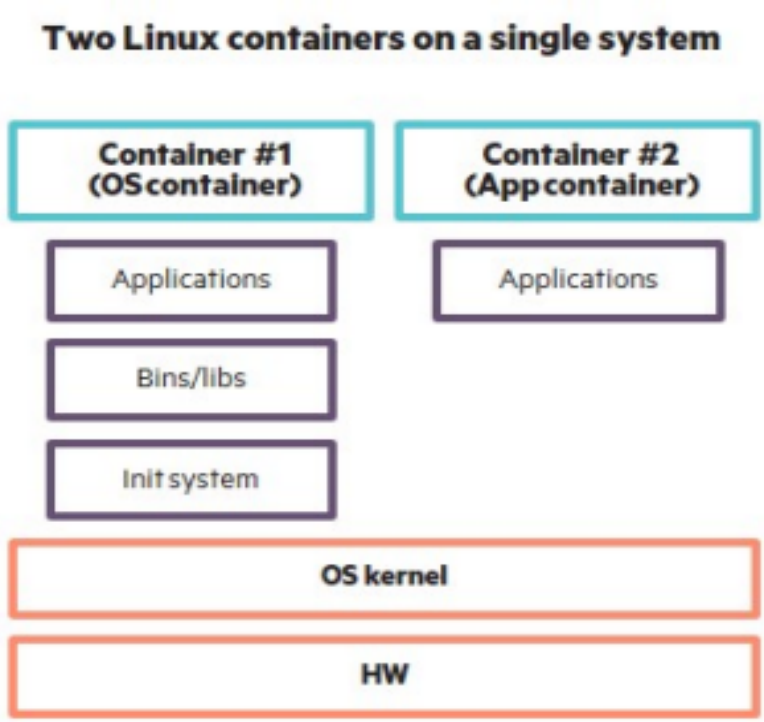
- Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes.
- The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers.
- This JSON is considered to be immutable, because changing it would change the computed ImageID.
- Changing it means creating a new derived image, instead of changing the existing image.

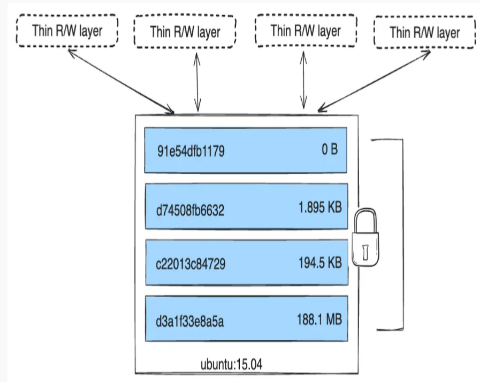
https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

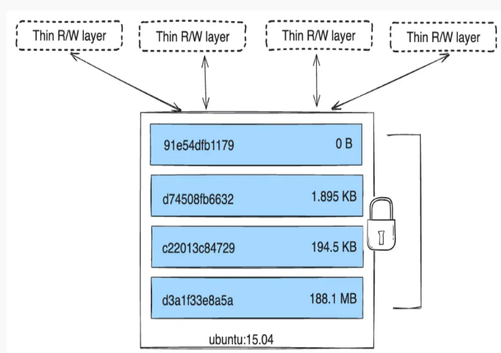
Containers only have access to resources that are defined in the image,

https://www.hpe.com/us/en/what-is/docker.html
</td></tr>
</table>

Claim 1	Accused Instrumentalities
	DESCRIPTION top The programs **ld.so** and **ld-linux.so*** find and load the shared objects (shared libraries) needed by a program, prepare the program to run, and then run it. https://man7.org/linux/man-pages/man8/ld.so.8.html
[1e] ii) wherein an instance of a SLCSE provided to at least a first of the plurality of software applications from the shared library is run in a context of said at least first of the plurality of software applications without being shared with other of the plurality of software applications and where at least a second of the plurality of software applications running under the operating system have use of a unique instance of a corresponding critical system element for performing same function, and	In each Accused Instrumentality, an instance of a SLCSE provided to at least a first of the plurality of software applications from the shared library is run in a context of said at least first of the plurality of software applications without being shared with other of the plurality of software applications and where at least a second of the plurality of software applications running under the operating system have use of a unique instance of a corresponding critical system element for performing same function. When a Docker or Kubernetes image is used to create a container, it creates a separate and isolated instance of a runtime environment which is independent of other containers running on the same host. Each container has its own instance of base images and its own data. The containers run in isolation, ensuring that the SLCSEs stored in the shared library are accessible to the software applications running in their respective containers. The image includes essential system files, libraries, and dependencies required to run the software application within the container. The containers can share common dependencies and components using layered images. This means that multiple containers utilize the same base image to create an instance. When an instance of SLCSE is provided from the base image (i.e., from the shared library) to an individual container including application software, it operates in isolation and runs its own instance of the software application without sharing resources or critical system elements with other containers. This ensures that each container has its own isolated context. Docker or Kubernetes containers can share common dependencies and components using layered images. This means that multiple containers can utilize the same base image. Therefore, each container, containing the application software running under the operating system, utilizes a unique instance of the corresponding critical system element to execute the respective application software for performing a same ~~or a different~~ function. *See, e.g.*:

Claim 1	Accused Instrumentalities
	Containers provide the core runtime abstraction for the user applications. These containers provide isolation between user applications and the rest of the infrastructure. The containers are based on Docker. https://support.hpe.com/hpesc/public/docDisplay?docId=a00097165en_us&docLocale=en_US&page=GUID-6B6676DB-AF5F-4555-B6AB-D2C11A89F320.html  https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/whitepaper/pdfs/4AA6-2761ENW.pdf

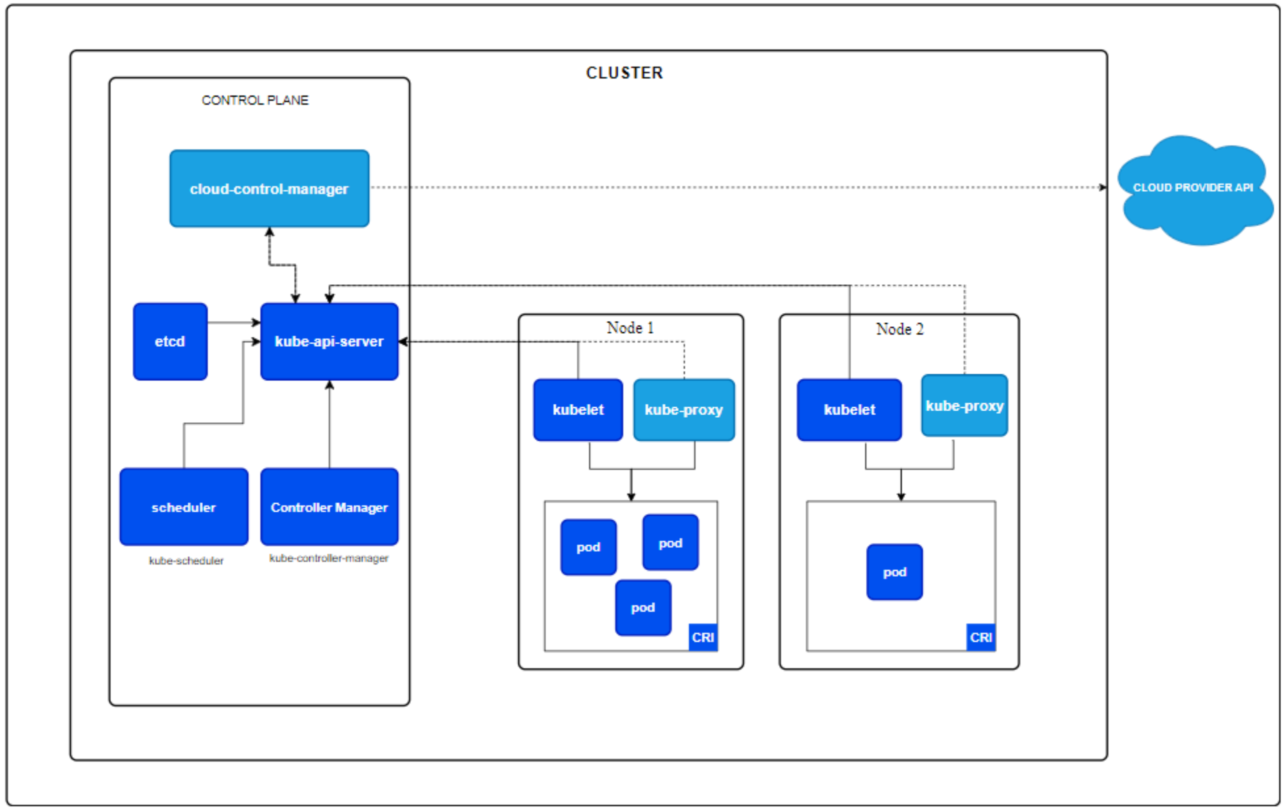
Claim 1	Accused Instrumentalities
	Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.  https://docs.docker.com/storage/storagedriver/ Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. https://www.techtarget.com/searchitoperations/definition/Docker-image
[1f] iii) wherein a SLCSE related to a predetermined function is provided to the first of the plurality of software applications for running a first instance of the SLCSE, and wherein a SLCSE for performing a same function is provided to the second of the plurality of software applications for running a second instance of the SLCSE simultaneously.	In each Accused Instrumentality, a SLCSE related to a predetermined function is provided to the first of the plurality of software applications for running a first instance of the SLCSE, and wherein a SLCSE for performing a same function is provided to the second of the plurality of software applications for running a second instance of the SLCSE simultaneously. For example, in Docker or Kubernetes containers, each container operates independently, and a base image includes essential system files, libraries, and dependencies (i.e., SLCSEs) required to run the software application within the container. Based on information and belief, each element, such as system files, libraries, and dependencies (i.e., SLCSE) is associated with an execution of a predetermined function related to the application. When an image is used to create a container in the Accused Instrumentality, an instance of the SLCSE is provided to a software application. Therefore,

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>
different instances of the SLCSE are provided to different applications for performing ~~either~~ a same ~~or a different~~ function, simultaneously.
See, e.g.:
Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container.
https://www.techtarget.com/searchitoperations/definition/Docker-image, Last accessed on June 14, 2023
A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.
https://docs.docker.com/get-started/overview/
Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.


https://docs.docker.com/storage/storagedriver/
</td></tr>
</table>

Claim 2

Claim 2	Accused Instrumentalities
2. A computing system as defined in claim 1, wherein in operation, multiple instances of an SLCSE stored in the shared library run simultaneously within the operating system.	Each Accused Instrumentality comprises or constitutes a computing system as defined in claim 1, wherein in operation, multiple instances of an SLCSE stored in the shared library run simultaneously within the operating system. For example, an individual host/node runs multiple containers and/or pods simultaneously, each of which has an instance of an SLCSE. When the multiple containers and/or pods run simultaneously, the multiple instances of the SLCSE stored in the shared library run simultaneously. *See, e.g.*:

Claim 2	Accused Instrumentalities
	 Kubernetes cluster architecture https://kubernetes.io/docs/concepts/architecture/

Claim 2	Accused Instrumentalities
	Containers Each container that you run is repeatable; the standardization from having dependencies included means that you get the same behavior wherever you run it. Containers decouple applications from the underlying host infrastructure. This makes deployment easier in different cloud or OS environments. Each node in a Kubernetes cluster runs the containers that form the Pods assigned to that node. Containers in a Pod are co-located and co-scheduled to run on the same node. https://kubernetes.io/docs/concepts/containers/

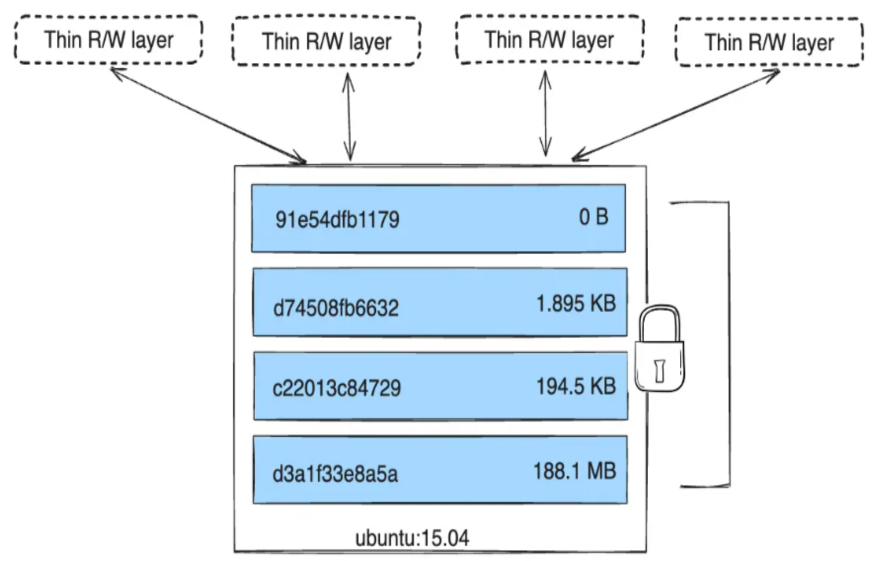
Claim 2	Accused Instrumentalities
	Kubernetes Scheduler In Kubernetes, *scheduling* refers to making sure that Pods are matched to Nodes so that Kubelet can run them. Scheduling overview A scheduler watches for newly created Pods that have no Node assigned. For every Pod that the scheduler discovers, the scheduler becomes responsible for finding the best Node for that Pod to run on. The scheduler reaches this placement decision taking into account the scheduling principles described below. If you want to understand why Pods are placed onto a particular Node, or if you're planning to implement a custom scheduler yourself, this page will help you learn about scheduling. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

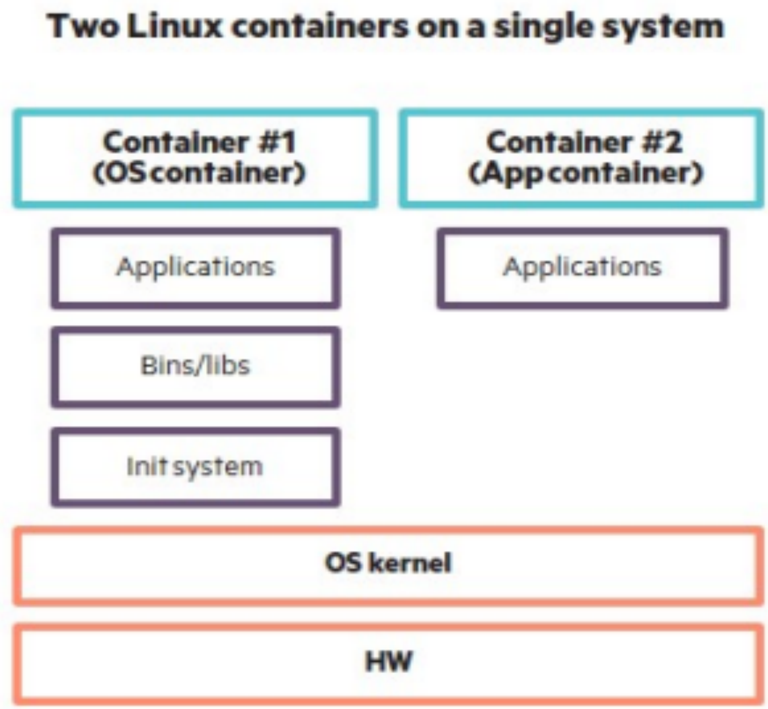
Claim 2	Accused Instrumentalities
	Running containers Docker runs processes in isolated containers. A container is a process which runs on a host. The host may be local or remote. When you execute `docker run`, the container process that runs is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host. https://docs.docker.com/engine/reference/run/

Claim 3

Claim 3	Accused Instrumentalities
3. A computing system according to claim 1 wherein OSCSEs corresponding to and capable of performing the same function as SLCSEs remain in the operating system kernel.	Each Accused Instrumentality comprises or constitutes a computing system according to claim 1 wherein OSCSEs corresponding to and capable of performing the same function as SLCSEs remain in the operating system kernel. For example, both Docker and Kubernetes systems preserve the host kernel substantially unchanged; therefore the OSCSEs corresponding to the SLCSEs remain in the operating system kernel. *See, e.g.*: Containers provide the core runtime abstraction for the user applications. These containers provide isolation between user applications and the rest of the infrastructure. The containers are based on Docker. https://support.hpe.com/hpesc/public/docDisplay?docId=a00097165en_us&docLocale=en_US&page=GUID-6B6676DB-AF5F-4555-B6AB-D2C11A89F320.html The container starts with a base image, and the microservice is packaged into a container image and then deployed through the container platform. The container platform is based on https://www.hpe.com/us/en/what-is/container-platform.html

Claim 3	Accused Instrumentalities
	## Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/ The idea of containerization is to isolate and package the application with all the dependencies in a container. https://community.hpe.com/t5/hpe-blog-uk-ireland-middle-east/containerization-the-next-generation-of-virtualization/ba-p/7154442 Container image files are complete, static and executable versions of an application or service and differ from one technology to another. Docker images are made up of multiple layers, which start with a base image that includes all of the dependencies needed to execute code in a container. Each image has a readable/writable layer on top of static unchanging layers. Because each container has its own specific container layer that customizes that specific container, underlying image layers can be saved and reused in multiple containers. An Open Container Initiative (OCI) https://www.techtarget.com/searchitoperations/definition/container-containerization-or-container-based-virtualization

Claim 3	Accused Instrumentalities
	Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.  https://docs.docker.com/storage/storagedriver/

Claim 3	Accused Instrumentalities
	 https://h50146.www5.hpe.com/products/software/oe/linux/mainstream/support/whitepaper/pdfs/4AA6-2761ENW.pdf

Claim 4

Claim 4	Accused Instrumentalities
4. A computing system according to claim 1 wherein the one or more SLCSEs provided to one of the plurality of software applications having exclusive use thereof, use system calls to access services in the operating system kernel.	Each Accused Instrumentality comprises or constitutes a computing system according to claim 1 wherein the one or more SLCSEs provided to one of the plurality of software applications having exclusive use thereof, use system calls to access services in the operating system kernel. For example, the SLCSEs in a container use system calls to access services in the operating system kernel. For example, the glibc library (or other similar library) in the container uses system calls to

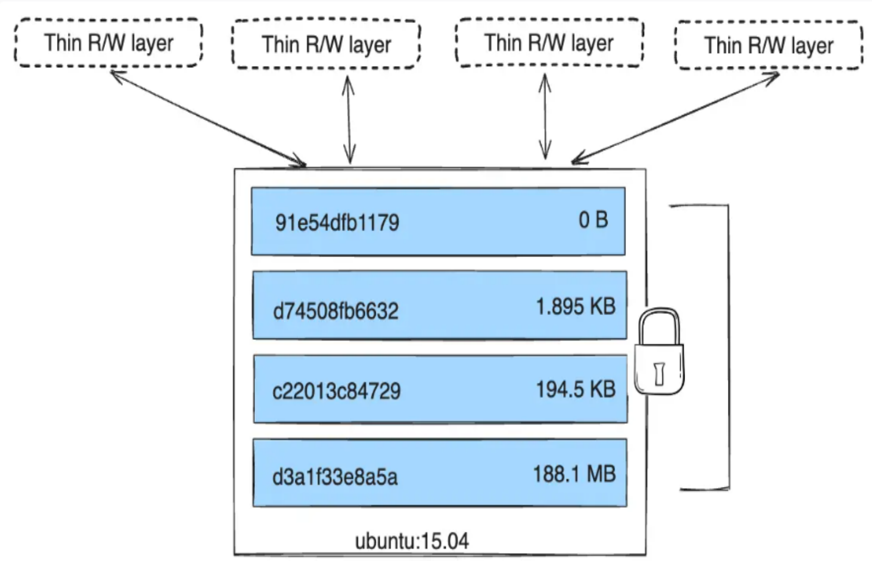
Claim 4	Accused Instrumentalities
	interface with the host Linux kernel. In general, system calls can be observed using a tool such as strace. *See, e.g.*: The **GNU C Library**, commonly known as **glibc**, is the GNU Project implementation of the C standard library. It is a wrapper around the system calls of the Linux kernel for application use. Despite its name, it now also directly supports C++ (and, indirectly, other programming languages). It was started in the 1980s by the Free Software Foundation (FSF) for the GNU operating system. https://en.wikipedia.org/wiki/Glibc

Claim 4	Accused Instrumentalities

	We can now get the process id directly from the cgroup. It will be located in the `cgroup.procs` file.
	`### Terminal 2 - Worker Node ###`
	`# Get the process id`
	`$ cat cri-containerd-ceeeef06afe89c8223d33b11e8d9e0b207118ac4dac3af826687668ee1ee`
	`16254`
	`# Validate what is running under the process`
	`$ ps aux | grep 16254`
	`azureus+    16254 0.0  0.1 713972 10476 ?       Ssl  15:04   0:00 ./faultyapp`
	`azureus+   94806  0.0  0.0   7004  2168 pts/0    S+   16:22   0:00 grep --color=a`
	Got it! With that, we can try to find out what is going out inside the app. Lets try to run `strace` to get some more insight.
	`### Terminal 2 - Worker Node ###`
	`$ sudo strace -p 16254 -f`
	`...`
	`# The app is trying to read a file port.txt`
	`[pid 16269] openat(AT_FDCWD, "port.txt", O_RDONLY|O_CLOEXEC <unfinished ...>`
	`[pid 16254] epoll_pwait(5,  <unfinished ...>`
	`# The file does not exist`
	`[pid 16269] <... openat resumed>)       = -1 ENOENT (No such file or directory)`
	`[pid 16254] <... epoll_pwait resumed>[], 128, 0, NULL, 0) = 0`
	`[pid 16269] write(1, "Something went wrong...\\n", 24 <unfinished ...>`
	After filtering the output, we can see the application is trying to read a text file called `port.txt`, and a few lines later, there is a message stating `ENOENT (No such file or directory)`. Let's create that file.
	https://www.berops.com/blog/a-different-method-to-debug-kubernetes-pods

Claim 18

Claim 18	Accused Instrumentalities
18. A computer system as defined in claim 2 wherein SLCSEs are not copies of OSCSEs.	Each Accused Instrumentality comprises or constitutes a computer system as defined in claim 2 wherein SLCSEs are not copies of OSCSEs. For example, in a typical case the SLCSEs come from a Linux distribution independent of the host operating system, and thus are not identical to the OSCSEs. ~~For another example, the SLCSEs are provided to the computer system through a separate process than the process by which the OSCSEs are provided to the computer system, and thus are not copied from the OSCSEs.~~ *See, e.g.*: Hewlett Packard Enterprise provides publicly available base OS images for use in containerized clusters. These images extend the base OS images available from Docker hub by adding several packages that permit HPE Ezmeral Runtime Enterprise to manage container orchestration seamlessly and to improve the security of the container. https://docs.ezmeral.hpe.com/runtime-enterprise/55/app-workbench-5-1/custom-base-images/AWB51_About_Custom_Base_Images.html The idea of containerization is to isolate and package the application with all the dependencies in a container. https://community.hpe.com/t5/hpe-blog-uk-ireland-middle-east/containerization-the-next-generation-of-virtualization/ba-p/7154442 Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/

Claim 18	Accused Instrumentalities
	Docker is used to create, run and deploy applications in containers. A Docker image contains application code, libraries, tools, dependencies and other files needed to make an application run. When a user runs an image, it can become one or many instances of a container. ttps://www.techtarget.com/searchitoperations/definition/Docker-image Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.  https://docs.docker.com/storage/storagedriver/ Containers only have access to resources that are defined in the image, https://www.hpe.com/us/en/what-is/docker.html

UNITED STATES DISTRICT COURT
FOR THE EASTERN DISTRICT OF TEXAS
MARSHALL DIVISION

VIRTAMOVE, CORP., Plaintiff, v. HEWLETT PACKARD ENTERPRISE COMPANY, Defendant.	Case No. 2:24-cv-00093-JRG-RSP

PLAINTIFF VIRTAMOVE, CORP.'S ~~CORRECTED~~ SUPPLEMENTAL PRELIMINARY DISCLOSURE OF ASSERTED CLAIMS AND INFRINGEMENT CONTENTIONS

I. Patent Rule 3-1: Disclosure of Asserted Claims and Infringement Contentions

Pursuant to Patent Rule 3-1, Plaintiff VirtaMove, Corp. submits the following Preliminary Disclosure of Asserted Claims and Infringement Contentions. This disclosure is based on the information available to VirtaMove as of the date of this disclosure, and VirtaMove reserves the right to amend this disclosure to the full extent permitted, consistent with the Court's Rules and Orders.

A. Patent Rule 3-1(a): Asserted Claims

VirtaMove asserts that Defendant Hewlett Packard Enterprise Company ("Defendant" or "HPE") infringes the following claims (collectively, "Asserted Claims"):

(1) U.S. Patent No. 7,519,814 ("the '814 patent"), claims 1, 2, 6, 9, and 10; and

(2) U.S. Patent No. 7,784,058 ("the '058 patent"), claims 1–4 and 18.

This Supplemental Preliminary Disclosure of Asserted Claims and Infringement Contentions includes the ~~This~~ Corrected Preliminary Disclosure of Asserted Claims and Infringement Contentions previously served. The Corrected Preliminary Disclosure of Asserted

Claims and Infringement Contentions correctly reflecteds, consistent with the Complaint (Dkt. 1) and First Amended Complaint (Dkt. 40), that the only independent claims that are asserted in this case are independent claim 1 of the '814 patent and independent claim 1 of the '058 patent. Independent Claim 31 of the '814 patent is not, was not, and will not be asserted in this case. Otherwise, ~~this~~ the Corrected Preliminary Disclosure of Asserted Claims and Infringement Contentions ~~is~~ was identical to the previously served Preliminary Disclosure of Asserted Claims and Infringement Contentions.

B. Patent Rule 3-1(b): Accused Instrumentalities of which VirtaMove is aware

VirtaMove asserts that the Asserted Claims are infringed by the various instrumentalities used, made, sold, offered for sale, or imported into the United States by Defendant, including certain (a) HPE products and services using secure containerized applications, including without limitation Ezmeral Runtime Enterprise (including without limitation both Ezmeral Runtime Enterprise and Ezmeral Runtime Enterprise Essentials, in each case including when marketed, sold, and/or licensed as part of or associated with HPE's GreenLake branding, e.g. "HPE GreenLake for containers" which "is built on HPE Ezmeral Container Platform") ~~and HPE GreenLake~~, and all versions and variations thereof since the issuance of the '814 patent; and (b) HPE products and services using user mode critical system elements as shared libraries, including without limitation Ezmeral Runtime Enterprise ~~–~~ (including without limitation both Ezmeral Runtime Enterprise and Ezmeral Runtime Enterprise Essentials, in each case including when marketed, sold, and/or licensed as part of or associated with HPE's GreenLake branding, e.g. "HPE GreenLake for containers" which "is built on HPE Ezmeral Container Platform")~~and HPE GreenLake~~, and all versions and variations thereof since the issuance of the '058 patent ("Accused Instrumentalities"). Defendant's Accused Instrumentalities of which VirtaMove is presently aware

are described in more detail in the accompanying preliminary infringement contention charts.

VirtaMove reserves the right to accuse additional products from Defendant to the extent VirtaMove becomes aware of additional products during the discovery process. Unless otherwise stated, VirtaMove's assertions of infringement apply to all variations, versions, and applications of each of the Accused Instrumentalities, on information and belief, that different variations, versions, and applications of each of the Accused Instrumentalities are substantially the same for purposes of infringement of the Asserted Claims.

C. Patent Rule 3-1(c): Claim Charts

VirtaMove's analysis of Defendant's products is based upon limited information that is publicly available, and based on VirtaMove's own investigation prior to any discovery in these actions. Specifically, VirtaMove's analysis is based on certain limited resources that evidence certain products made, sold, used, or imported into the United States by Defendant.

VirtaMove reserves the right to amend or supplement these disclosures for any of the following reasons:

(1) Defendant and/or third parties provide evidence relating to the Accused Instrumentalities;

(2) VirtaMove's position on infringement of specific claims may depend on the claim constructions adopted by the Court, which has not yet ~~occured~~occurred; and

(3) VirtaMove's investigation and analysis of Defendant's Accused Instrumentalities is based upon public information and VirtaMove's own investigations. VirtaMove reserves the right to amend these contentions based upon discovery of non-public information that VirtaMove anticipates receiving during discovery.

Attached and incorporated herein in their entirety, are charts identifying where each element of the Asserted Claims are found in the Accused Instrumentalities.

Unless otherwise indicated, the information provided that corresponds to each claim element is considered to indicate that each claim element is found within each of the different variations, versions, and applications of each of the respective Accused Instrumentalities described above.

D. Patent Rule 3-1(d): Literal Infringement / Doctrine of Equivalents

With respect to the patents at issue, each element of each Asserted Claim is considered to be literally present. VirtaMove also contends that each Asserted Claim is infringed or has been infringed under the doctrine of equivalents in Defendant's Accused Instrumentalities. VirtaMove also contends that Defendant both directly and indirectly infringes the Asserted Claims. For example, the Accused Instrumentalities are provided by the Defendant to customers, who are actively encouraged and instructed (for example, through Defendant's online instructions on its website and instructions, manual, or user guides that are provided with the Accused Instrumentalities) by Defendant to use the Accused Instrumentalities in ways that directly infringe the Asserted Claims. Defendant therefore specifically intends for and induces its customers to infringe the Asserted Claims under Section 271(b) through the customers' normal and customary use of the Accused Instrumentalities. In addition, Defendant is contributorily infringing the Asserted Claims under Section 271(c) and/or Section 271(f) by selling, offering for sale, or importing the Accused Instrumentalities into the United States, which constitute a material part of the inventions claimed in the Asserted Claims, are especially made or adapted to infringe the Asserted Claims, and are otherwise not staple articles or commodities of commerce suitable for non-infringing use.

E. Patent Rule 3-1(e): Priority Dates

The Asserted Claims of the '814 patent are entitled to a priority date at least as early as September 15, 2003, the filing date of provisional application No. 60/502,619.

The Asserted Claims of the ’058 patent are entitled to a priority date at least as early as September 22, 2003, the filing date of provisional application No. 60/504,213.

A diligent search continues for additional responsive information and VirtaMove reserves the right to supplement this response.

F. Patent Rule 3-1(f): Identification of Instrumentalities Practicing the Claimed Invention

At this time, VirtaMove does not identify any of its instrumentalities as practicing the Asserted Claims. A diligent search continues for additional responsive information and VirtaMove reserves the right to supplement this response.

II. Patent Rule 3-2: Document Production Accompanying Disclosure

Pursuant to Patent Rule 3-2, VirtaMove previously submitted the following Document Production Accompanying Disclosure, along with an identification of the categories to which each of the documents corresponds.

F. Patent Rule 3-2(a) documents:

VirtaMove is presently unaware of any documents sufficient to evidence any discussion with, disclosure to, or other manner of providing to a third party, or sale of or offer to sell, the inventions recited in the Asserted Claims of the Asserted Patents prior to the application dates or priority dates for the Asserted Patents. A diligent search continues for such documents and VirtaMove reserves the right to supplement this response.

G. Patent Rule 3-2(b) documents:

VirtaMove identifies the following non-privileged documents as related to evidencing conception and reduction to practice of each claimed invention of the Asserted Patents: VM_HPE_0000865–VM_HPE_0000880. A diligent search continues for additional documents and VirtaMove reserves the right to supplement this response.

H. Patent Rule 3-2(c) documents:

VirtaMove identifies the following documents as being the file histories for the Asserted Patents: VM_HPE_0000001–VM_HPE_0000864.

Dated: ~~July 1~~September 6, 2024

Respectfully submitted,

/s/ Reza Mirzaie
Reza Mirzaie
CA State Bar No. 246953
Marc A. Fenster
CA State Bar No. 181067
Neil A. Rubin
CA State Bar No. 250761
Amy E. Hayden
CA State Bar No. 287026
Jacob R. Buczko
CA State Bar No. 269408
James S. Tsuei
CA State Bar No. 285530
James A. Milkey
CA State Bar No. 281283
Christian W. Conkle
CA State Bar No. 306374
Jonathan Ma
CA State Bar No. 312773
Daniel Kolko (CA SBN 341680)
RUSS AUGUST & KABAT
12424 Wilshire Boulevard, 12th Floor
Los Angeles, CA 90025
Telephone: 310-826-7474
Email: rmirzaie@raklaw.com
Email: mfenster@raklaw.com
Email: nrubin@raklaw.com
Email: ahayden@raklaw.com
Email: jbuczko@raklaw.com
Email: jtsuei@raklaw.com
Email: jmilkey@raklaw.com
Email: cconkle@raklaw.com
Email: jma@raklaw.com
Email: dkolko@raklaw.com

Qi (Peter) Tong
4925 Greenville Ave., Suite 200

Dallas, TX 75206
Email: ptong@raklaw.com

ATTORNEYS FOR PLAINTIFF VIRTAMOVE, CORP.

CERTIFICATE OF SERVICE

I certify that this document is being served upon counsel of record for Defendants on ~~July 1~~September 6, 2024 via e-mail.

/s/ Reza Mirzaie
Reza Mirzaie